PATENT APPLICATION
Attorney Doc. No. 3222-5

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s):	Sanjay Verma et al.	Confirmation No.	5357
Serial No.	10/696,416	Examiner:	Monica M. PYO
Filed:	October 28, 2003	Group Art Unit:	2161

For: METHOD AND APPARATUS FOR INCREASING TRANSACTION CONCURRENCY BY EARLY RELEASE OF LOCKS IN GROUPS

Box Non-Fee Amendment
Assistant Commissioner for Patents
Washington, D.C. 20231

DECLARATION OF SANJAY VERMA UNDER RULE 37 C.F.R. 1.131

I, Sanjay Verma, declare the following:

1. I am a co-inventor of the subject matter described in the pending patent application titled: METHOD AND APPARATUS FOR INCREASING TRANSACTION CONCURRENCY BY EARLY RELEASE OF LOCKS IN GROUPS, Ser. No. 696,416, filed October 28, 2003.

2. I previously worked for TIMES TEN PERFORMANCE SOFTWARE, 800 West El Camino Real, Mountain View, California 94040, the previous assignee of the present patent application.

3. Before April 8, 2003, I and the other named inventors of U.S. Serial No. 10/696,416 conceived of an Activity Duration Locking (ADL) scheme that allows a transaction fine control over creation of lock groups and classification of locks into lock groups. The life time that the lock group is active defines a new lock duration. This allows lock durations to be defined with the creation of a new lock group and any number of lock durations can be active for a given transaction.

4. Attached as Exhibit A is a document that was written prior to April 8, 2003, where I and the other named inventors of the 10/696,416 patent application describe our invention. The disclosure in Exhibit A describes how lock durations can be associated with different activities in a transaction and locks maintained for the duration of the activities and then the locks released when the activities are completed.

5. Attached as Exhibits B and C are lists of the code that was written by me and the other co-inventors that implements the Activity Duration Lock described in Exhibit A. The code listed in Exhibits B and C was written in the United States prior to April 8, 2003 and associates lock durations with different activities in a transaction and maintains locks for the duration of the activities and then releases the locks when the activities are completed. The bottom of Exhibit B shows an implemented lock compatibility matrix that encodes the compatibility rules for the multiple different lock modes that are maintained on data items for the duration of associated activities and then released when the activities are completed. Exhibit C lists the routines

implemented in the code for Activity based locking and Activity related code (Group locking) that was checked in prior to April 8, 2003.

6. The Bhattacharjee et al. reference (US 2004/0205066 A1) was used to reject the claims of our application under 35 U.S.C. § 102(e). Applicants wish to "swear behind" this reference. Although Bhattacharjee et al has an effective filing date of April 8, 2003 that precedes the present application's effective filing date of October 28, 2003, Applicants conceived and reduced the invention to practice prior to April 8, 2003.

I, the undersigned, declare that all statements made herein of my own knowledge are true, and that all statements made on information and belief are believed to be true; and further, that these statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United Stated Code, and that such willful false statements may jeopardize the validity of the application of any patent issuing thereon.

DATED this 4th day of September, 2006.

Sanjay Verma

Sanjay Verma

tabbies®
EXHIBIT
A

Activity duration lock - a technique to increase transaction concurrency by early releasing of locks in groups

INVENTORS (in alphabetical order)

Sibsanker Haldar, Sherry Listgarten, Sanjay Verma

ASSIGNEE

TimeTen Performance Software, Mountain View, CA

ABSTRACT

A database management system (DBMS, for short) supports transactions to access data items stored in a database. Before actually accessing the items, a transaction acquires locks on the data items that it wishes to access. When the transaction does not require any further access to the said data items, the transaction may release the locks it had acquired on the data items. Thus a lock is acquired and held on a data item for the duration for which the transaction wishes to own access permissions on the given data item. This duration is commonly termed as *lock duration*. A reference is made to [JimGray93], Chapter 8, Lock Implementation, for a further discussion of lock durations. Commercially available DBMSs use a few meaningful lock durations known a priori. Some commonly known lock durations are instant, short, medium, long, etc. All the myriad needs of access permissions that a transaction may have are grouped under this known finite number of durations. There are various cases where the finite set of known durations is inadequate to satisfy the access needs of a transaction. This becomes especially important in the case of non-serializable transactions.

The concept of division of a transaction into various tasks via the means of sub transactions and multi level systems is well known in the current art. A reference is made to [DavidLomet92], [Weikum86], [Weikum91] for complete discussion on sub transactions and multi level concurrency control. However, concept of sub-transactions and multi level transactions are complex concepts and expensive implementations for what can be satisfied with a simple concept of lock grouping. A finite set of lock durations, does not provide for groupings of locks in sets that are unknown a priori. We propose the concept of *activity duration* as a lightweight solution for lock groupings. Activity duration locking scheme, described here where transactions may have fine control over selection of lock durations to enable faster releasing of locks in groups, provides a superior alternative to existing locking schemes. Activity duration locking scheme supports a very general, somewhat abstract, concept of lock durations without any a priori semantics attached to individual lock durations.

FIELD OF INVENTION

The invention relates to the field of data processing systems. More specifically, the invention relates to the locking techniques for management of data in database systems. More particularly, the invention concerns durations of locks acquired on various data records and release of locks in groups.

BACKGROUND OF INVENTION

Database Management System (DBMS), Lock Manager (LM)

A database is a collection of related data items. To perform various tasks, users, via transactions, access data items stored in the database. These user accesses need to be controlled to ensure data consistency. A database management system is a software package that supports means for controlling database access, and thus maintaining data consistency. This is usually in the form of transactions, which is a grouping of various activities into one unit. A reference is made to [JimGray93], Chapter 4, Transaction Models, for further discussion on the concept of transactions.

When a transaction is executed in a database, the transaction may read and write data items from the database. To get a consistent view of the data items, the transaction acquires locks on the data items before accessing them. The transaction may release these locks when it no longer needs access permissions to the data item. By default, all locks that the transaction owns are released when the transaction completes. Thus it can be inferred that every lock that the transaction acquires has an associated life-span. This life-span is the time elapsed between when the lock is acquired and when the lock is released. The lock life-span, which is popularly referred to as *lock duration* in the literature, is typically shorter than the life span of the lock acquiring transaction. Lock manager, an important module of a DBMS processes locking and unlocking requests by transactions. LM should also support the ability to release locks in a given group before the transaction finishes, if there is such a need.

Transactions have an additional property called isolation. A reference is made to [JimGray93], Chapter 7, Isolation Concepts, for further discussion on typical isolation levels and degrees of isolation. Generally DBMSs support serializable, read committed, dirty read etc. isolation levels. For most non serializable isolation levels, locks may be released on data items read before the transaction finishes. One such example is read committed isolation level. According to the consistency semantics of this isolation level, the most recently committed value of the given data item should be read by the accessing transaction. One implication of this is that the transaction may request a lock on, read and then release the lock on a given data item, and this may be repeated as many times as desired, where each read of the same data item may potentially return a different value. Such a transaction may need to group locks and release them in one call to LM, instead of releasing all such locks individually.

Commercially available DBMSs support a few lock duration. Each duration has a distinct name and predefined semantics. For example, instant, short, medium and long lock durations are used in many commercially available DBMSs. Note that the duration of a lock is not determined by LM, but by the transaction that requests the lock. A transaction may, based on its isolation level, choose different durations for the same data item. If the transaction is executing under serializable isolation, then the lock durations are trivially from the time acquired to the end of the transaction. In this case, all locks of a transaction are released when the transaction execution is complete.

However, for weaker isolations (Cursor stability, read-committed), lock durations play a very important role. Transactions may request release for some locks before transactions the are complete. Such early release of locks is necessary to promote transaction concurrency. Reducing lock duration reduces access conflicts. For example, a transaction may request locks that are to be released early in the medium duration. When the transaction determines that access is no longer needed to any data item in that set, it may request the LM to release all medium duration locks.

As is clear, the mechanism as described is limited in functionality, since there is no meaningful way to define and manage potentially unlimited number of groups, into which a transaction may want to classify a given lock. Managing these groups with a few a priori meaningful lock durations will necessitate a separate queue manager that will need to note all classifications; this proves to be a cumbersome and complicated task. To provide an efficient simple solution to manage a potentially large number of concurrently active lock groups, we define "*activity duration locking*".

In activity duration locking scheme, which is a superior alternative to existing locking schemes, a transaction has fine control over creation of lock groups and classification of locks into lock groups. The life time for which the lock group is active defines a new lock duration. Thus large number of lock durations can be defined via creation of new lock groups, and any number of lock durations can therefore be active for a given transaction. The scheme supports a very general concept of lock duration without any a priori semantics to individual lock durations. Duration may be determined by the transaction as desired.

DETAILS OF INVENTION

It should be noted that the concept of finite number of lock durations and their associated semantics is limited. Associating names with lock durations create mental bottleneck to thinking pattern, since every name is associated with a pre determined form and meaning. Additionally, only a few lock durations may not be suitable to effectively manipulate lock usage for complex non serializable transactions. Thus, we need to broaden the notion of the lock duration by enlarging its cardinality, and in such a scheme it is not meaningful to name with durations. Activity lock duration scheme, in theory, supports infinitely many unnamed lock durations.

A transaction may be viewed at different levels of abstractions. At the lowest level, it is a set of reads and writes on data items which are stored in a database. At the intermediate level, a transaction is viewed as a collection of *activities*. An activity is a set of closely related actions on data items. That is, an activity is a set of transaction's reads and writes, and in turn is a subset of all actions performed by the transaction. For example, compilation of a SQL statement is an activity. Any transaction, as a whole, is itself an activity.

We view each and every transaction as a collection of distinct activities, which as described are sets of lower level reads and writes on data items. Each activity spans a finite period in time. Activities of a transaction may overlap in time, that is, they may be concurrent. Each activity accesses a finite set of data items, and every data accessed by the transaction belongs to a finite set of activities (Note the similarity between this definition and the definition of *higher level view* as described in [Lamport 86]). A transaction locks data items through different activities. In theory, a transaction can have arbitrary many concurrent activities. An activity may lock an arbitrary number of data items, and these locks are to be released when the activity finishes.

We must have efficient means of classifying locks into activities with minimum overhead and also of releasing all locks in an activity, when the activity completes. The currently available support for lock durations does not satisfy this need, since first there is no concept of an arbitrary duration, and secondly the transactional component of DBMS will have to maintain a separate list of locks, instead of the LM managing that for the transactional component. Consequently it is complex and inefficient to keep track of association of locks to activities using ordinary lock durations.

Our new scheme provides an efficient and a simple solution to this need that the transactional component of a DBMS may have. We provide an efficient means of classification and grouping in the lock manager, and thus the transaction manager needs to note only the currently alive activity identifiers for the transactions. When a transaction starts a new activity, it chooses a unique and unused token for uniquely identifying the activity. This token will be referred to as activityID. The LM may help in determining the unique activityID. When the transaction manager request LM for a lock, activityID is sent in so that LM can classify the said lock under said activity. When it is determined that the activity is complete, the transaction asks LM to release locks associated with the said activity, identified via the unique activityID. This activityID is then declared free to be used by any new activities of the transaction. The LM releases all locks for the given activity together. Thus it can be clearly seen that all locks associated with this activity will be of this "activity duration". Thus arbitrarily many lock durations can be defined and maintained in a non-serializable transaction by the transactions themselves. The onus of defining lock duration now moves from the LM to the transaction manager. This provides additional flexibility in defining and managing any number of desired lock durations.

The activity duration locking scheme discussed above is implemented in TimesTen Performance Software product TimesTen 4.5. It has enabled a conceptual framework and a simple and efficient implementation in which to define lock durations, enabling early releasing of locks for non serializable transactions.

We introduce here an example where the notion of activities helps in efficiently completing the given task. Consider the notion of pre-fetching. When a transaction request access of multiple tuples, then the implementation choices are to retrieve these tuples from the database one by one, or to get these tuples in bulk. We consider this issue in the context of ODBC SQLFetch call. A reference is made to the ODBC interface [ODBC SQL3.0]. In such an environment the application typically provides space for only one tuple into which the retrieved data is to be copied. Thus the intermediate layer can retrieve each tuple one by one copy the data into the application buffer and return to the application. Another choice is to retrieve the tuple IDs in a bulk, and after the first fetch simply copy data from the current tupleID into the application buffer. This pre-fetching typically proves to be faster than retrieving each tuple one by one. For such a scheme, it becomes necessary to hold lock on each tuple in the pre-fetch group since modification to the pre-fetched tuples should be prohibited until data has been copied from the current tuple group. Again it is faster to unlock all tuples in a prefetch group in one operation rather than unlocking the tuples one by one. This need is ideally satisfied by the notion of activities. Before prefetching a group of tuples, the transaction starts an activity and asks the lock manager to associate the remaining locks in the prefetch group with that activity. After data from the last tuple has been copied into the application buffers, the transaction may request LM to release all locks pertaining to said activity. This solution is much more efficient than maintaining a list of locks obtained in the given prefetch session in the transaction and then finally unlocking these locks one by one.

Thus it can be seen that the notion of activities has proven to be very valuable in implementation of various optimizations and efficiently conforming to semantics of various isolation levels in the TimesTen DBMS. The contribution of the current invention is the notion of activities, which provides transaction managed durations. This model is very flexible.

References

[DavidLomet92] MLR: A Recovery Method for Multi-level Systems, David Lomet, ACM SIGMOD 1992
[JimGray93] Transaction Processing, Concepts and Techniques, J. Gray, A. Reuter, Morgan Kaufmann, 1993
[Lamport86] On Interprocess Communication. Part I: Basic Formalism and Part II: Algorithms, Leslie Lamport, Distributed Computing 1(2): 77-101, 1986
[Weikum86] A theoretical foundation of multi-level concurrency control, G. Weikum, Proc. Of ACM PODS Conf., March 1986
[Weikum91] Principles and realizations strategies of multilevel transaction management, ACM TODS 16,1, 1991

EXHIBIT

B

```
1.1          (shaldar          ): /*

1.1          (shaldar          ):  * The following matrix shows the compatibility between
various lock modes
1.1          (shaldar          ):  * that may be granted on data items to different
transactions simultaneously.
1.1          (shaldar           :  * The held mode is the supremum of all lock modes that
are now active
1.1          (shaldar          ):  * on the data. The matrix determines if a requested
mode can be granted
1.1          (shaldar           :  * immediately; or, the request needs to wait until
some non compatible
1.1          (shaldar           :  * locks are released or downgraded.
1.1          (shaldar           :  *
1.1          (shaldar          ):  * There are presently three data granules supported by
Timesten:
1.1          (shaldar           :  *      (1) database, (2) tables, and (3) tuples.
1.1          (shaldar           :  * Not all lock modes are used for all data granules.
1.1          (shaldar           :  * The following lists are the permitted ones.
1.1          (shaldar          ):  * D = {IS, IX, X} mode locks can be held on the
database
1.1          (shaldar           :  * T = {IS, S, IX, SIX, U, X} mode locks can be held on
tables
1.1          (shaldar              * R = {S, Sn, U, X, Xw, XNi, Un, En}  mode locks can
be held on rows/tuples.
1.1          (shaldar           :  *
1.1          (shaldar           :  * The valid entries in the Compat matrix are those of
1.1          (shaldar           :  * cross-product(D,D) union cross-product(T,T) union
cross-product(R,R).
1.1          (shaldar          ):  * Other associations are invalid. For the sake of
safety,
1.1          (shaldar          ):  * invalid associations are made incompatible.
1.1          (shaldar              *
1.1          (shaldar          ):  * Special note: A transaction can write a data item
only if it holds a X or Xw
1.1          (shaldar              *               mode lock. It cannot modify a data
item locked in
1.1          (shaldar           ;  *               Update mode: it has to get an X- or
Xw-mode lock
1.1          (shaldar           :  *               to modify the data.
1.1          (shaldar           :  *
1.1          (shaldar           :  * When dereferencing the Compat matrix, do the
following:
1.1          (shaldar           :  * Compat [ requested mode ] [ held mode ]
1.1          (shaldar          ):  */
1.1          (shaldar          ): static const sb_boolean_t Compat[sbLM_MAX][sbLM_MAX] =
{
1.1          (shaldar           :              /* Already held mode */
1.74         (verma            ): /*NULL  IS  IRC S   Sn  IX  SIX IU   U   X   W   Xw
XNi Un  En*/
1.74         (verma             :   {1,    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
1,  1,  1   /* NULL*/},
1.74         (verma            ):   {1,    1,  1,  1,  0,  1,  1,  1,  0,  0,  0,  0,
0,  0,  0   /* IS  */},
1.74         (verma             :   {1,    1,  1,  1,  1,  1,  1,  1,  1,  0,  1,  1,
1,  1,  1   /* IRC */},
1.74          (verma                    {1,     1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,
0,  1   /* S   */},
1.74          (verma                    {1,     0,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  1,
0,  1   /* Sn  */},
1.74          (verma                    {1,     1,  1,  0,  0,  1,  0,  1,  0,  0,  0,  0,  0,
0,  0   /* IX  */},
1.74          (verma                ):  {1,     1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,  0   /* SIX */},
1.74          (verma                 :  {1,     1,  1,  1,  0,  1,  1,  1,  0,  0,  0,  0,  0,
0,  0   /* IU  */},
1.74          (verma                    {1,     1,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  0,
0,  1   /* U   */ /* Requested mode */},
1.74          (verma                    {1,     0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,  0   /* X   */},
1.74          (verma                 :  {1,     0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,  0   /* W   */},
1.74          (verma                ):  {1,     0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,
0,  1   /* Xw  */},
1.74          (verma                ):  {1,     0,  1,  0,  1,  0,  0,  0,  0,  0,  0,  1,  1,
1,  1   /* XNi */},
1.74          (verma                 :  {1,     0,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,  1,
0,  1   /* Un  */},
1:74          (verma                ):  {1,     0,  1,  1,  1,  0,  0,  0,  1,  0,  0,  1,  1,
1,  1   /* En  */}
1.1           (shaldar              ): };
```

```
1.74         (verma              ):  * Revision 1.73                    verma
1.74         (verma              ):  * Allow sbLockENQ to populate the lockMgrInfo
data for all lock
1.74         (verma              ):  * requests. Now at sbLockActivity convert time,
we check if
1.74         (verma              ):  * the lock is read only. Now we can collect
lockMgrInfo even for
1.74         (verma              ):  * X locks.
1.74         (verma              ):  *
1.74         (verma              ):  * probTrak:
1.74         (verma              ):  *
1.53         (kirke              ):  * Revision 1.52                     kirke
1.53         (kirke              ):  * Corrected return value for
sbLockActivityConvert()
1.53         (kirke              ):  * probTrak: none
1.53         (kirke              ):  *
1.46         (mlm                ):  * Revision 1.45                     kirke
1.46         (mlm                ):  * Rewrote InternalLockLatch macros.
1.46         (mlm                ):  * probTrak: none
1.46         (mlm                ):  *
1.46         (mlm                ):  * Revision 1.44.2.4                      verma
1.46         (mlm                ):  * Add storage for current activity ID, if the
lock is indeed stored
1.46         (mlm                ):  * in an activity.  This is needed by joins to
function properly.
1.46         (mlm                ):  * probTrak:
1.46         (mlm                ):  *
1.16         (feldhaus           ):  * Revision 1.15                     verma
1.16         (feldhaus           ):  * New and Improved Activity Locking!
1.16         (feldhaus           ):  * All sbLockENQ signature changed.
1.16         (feldhaus           ):  * probTrak: 9700
1.16         (feldhaus           ):  *
1.12         (verma              ):  * Revision 1.11                     verma
1.12         (verma              ):  * Allow En locks to be taken under activities if
so desired.
1.12         (verma              ):  * probTrak:
1.12         (verma              ):  *
1.10         (shaldar            ):  * Revision 1.9                     verma
1.10         (shaldar            ):  * Make sure only read locks have any activity
associated with them.
1.10         (shaldar            ):  * probTrak: 9790
1.10         (shaldar             :  *
1.7          (shaldar            ):  * Revision 1.6                     verma
1.7          (shaldar             :  * 1. Set the activity Id in sbLockENQ
1.7          (shaldar            ):  * 2. Fix lathcing in sbLockDEQGroup
1.7          (shaldar            ):  * probTrak:9700
1.7          (shaldar            ):  *
1.4          (verma              ):  * Revision 1.3                     shaldar
1.4          (verma              ):  * Basic framework for activity based locking
(for Early Lock Relese 4.5 feature) is setup
1.4          (verma                  *
```

EXHIBIT
tabbies®
C